操作系统课程设计报告

姓　　名：　　李嘉鹏

学　　院：　　计算机科学与技术学院

专　　业：　　数据科学与大数据技术

班　　级：　　大数据 2101 班

学　　号：　　U202115652

指导教师：　　周正勇

分数	
教师签名	

2024 年 4 月 6 日

目　录

实验一 ChallengeX

1.1 实验目的

融合此前全部的挑战实验知识点（前四章 traps、memory、process 和 filesystem 的 Challenge1~Challenge3），并实现一个类似于 Bash 或 Linux Shell 的交互式 Shell。最终需要有命令证明此前的挑战实验已经融合进去。

1.2 实验内容

为了实现一个交互式 Shell，需要完成三部分：（1）实现 Shell 的交互功能，即读取用户输入流；（2）完成 Shell 的主程序；（3）将此前的挑战实验融合并增添加分项的功能。

下面将逐一介绍各部分的设计思路与关键代码。

1.2.1 实现 Shell 交互功能

首先，为了读取并解析用户的指令，需要写一个格式化字符串输入函数 customFormattedScan。与 scanf 函数类似，该函数可以从 stdin 中按照指定的格式提取数据，具体处理流程为：

（1）遍历格式字符串 formatString 和输入字符串 inputString，逐个字符进行匹配和处理。由于 Shell 接受用户的输入只可能为字符串，也就是用户程序的名称，而不涉及到单个存在的数字等，因此这里只需要处理格式为“%s”的输入。

（2）循环内部使用一个 switch 语句来处理不同的状态。开始时状态为 STATE_INITIAL，表示初始状态。如果遇到%字符，则进入 STATE_CONVERSION 状态，表示开始进行格式转换；如果遇到空白字符则跳过；否则，比较格式字符串和输入字符串中的字符是否相同。

（3）在 STATE_CONVERSION 状态下，对于%s 转换，从输入字符串中读取一个字符串，并根据给定的字段宽度进行处理。转换完成后，状态重新设置为 STATE_INITIAL。

（4）最后，函数返回成功解析的参数数量。由此函数即可实现读取用户输入流的功能，将读入的字符串进行格式化解析后作为参数传给相应位置。

函数代码如下所示。

```
int customFormattedScan(const char *inputString, const char *formatString, va_list
```

```
argumentsList)
{
    size_t count = 0;
    int state = STATE_INITIAL;
    void *dataPtr;
    int radix, isNegative, formatFlags, fieldWidth = 0, lengthFlags;

    if (!formatString)
        return 0;

    for (; *formatString && *inputString; formatString++)
    {
        switch (state)
        {
            case STATE_INITIAL:
                if (*formatString == '%')
                {
                    formatFlags = 0;
                    state = STATE_CONVERSION;
                }
                else if (isWhitespace(*formatString))
                    while (isWhitespace(*inputString))
                        inputString++;
                else
                    if (*formatString != *inputString++)
                        break;
                continue;
            case STATE_CONVERSION:
                if ('s' == *formatString)
                {
                    state = STATE_INITIAL;
                    if (formatFlags & FLAG_SKIP)
                    {
                        skipString(&inputString, NULL, NULL, fieldWidth);
                        continue;
                    }
                    dataPtr = va_arg(argumentsList, void *);
                    readString(&inputString, (char *)dataPtr, NULL, fieldWidth);
                    count++;
                }
                else if ('%' == *formatString)
                {
                    state = STATE_INITIAL;
                    if (*inputString != '%')
```

```
                    break;
                inputString++;
            }
            else
                break;
        }
    }

    return count;
}
```

1.2.2 完成 Shell 主程序

Shell 主程序为 user 文件夹下的 app_shell.c。基本思路是：

（1）用户一旦打开 Shell 即不会主动退出（除非输入“leave”），因此需要不断读入用户输入；

（2）若用户命令为“cd xxx”，则将当前工作目录 cwd 切换为用户指定的目录；

（3）当接收到一个以“\n”结尾的输入字符串且并非 leave 或 cd 命令时，创建一个进程并调用 exec 函数执行用户程序，主进程将会保持等待状态，直到创建的子进程执行完成并退出。

代码如下所示。

```
#include "user_lib.h"
#include "string.h"
#include "util/types.h"

int main(int argc, char *argv[])
{
  char cmd[64];
  char para[64];
  char cwd[64];
  printu("RISCV-PKE Shell. Type 'leave' to leave.\n");
  while (1)
  {
    read_cwd(cwd);
    printu("Shell >>> %s$ ", cwd);
    user_scan("%s %s", &cmd, &para);
    if (strcmp("leave", cmd) == 0)
      break;
    else if (strcmp("cd", cmd) == 0)
    {
```

```
      change_cwd(para);
      continue;
    }

    int pid = fork();
    if (pid == 0)
    {
      exec(cmd, para);
      return 1;
    }
    else
      wait(pid);
  }
  exit(0);
  return 0;
}
```

1.2.3 挑战实验运行结果与加分项

Shell 支持的全部命令（即用户程序名）及对应的挑战实验如表 1 所示。

表 1：Shell 支持的全部命令

用户命令（用户程序名）	功能	对应挑战实验
app_print_backtrace	打印用户程序调用栈	lab1_challenge1
app_errorline	打印异常代码行	lab1_challenge2
（无特定命令）	多核启动及运行	lab1_challenge3
app_sum_sequence	处理复杂缺页异常	lab2_challenge1
app_singlepageheap	堆空间管理	lab2_challenge2
（无特定命令）	多核内存管理	lab2_challenge3
app_wait	进程等待和数据段复制	lab3_challenge1
app_semaphore	进程信号量控制	lab3_challenge2
app_cow	进程写时复制	lab3_challenge3
cd app_pwd	相对路径（切换或打印当前路径）	lab4_challenge1
app_mkdir app_touch app_echo app_cat app_ls	创建目录 修改文件 向指定文件写入 hello world 读取文件 显示目录下的文件	lab4_challenge2 lab4_challenge3

下面简单展示表 1 中涉及到的各条用户命令的输出结果，如图 1~图 15 所示。据此可知 Shell 已经实现前面全部的挑战实验。

```
Shell >>> /bin$ app_print_backtrace
into wait
hartid = 0, Application: /bin/app_print_backtrace
hartid = 0, Application program entry point (virtual address): 0x000000000001011a
back trace the user app in the following:
f8
f7
f6
f5
f4
f3
f2
hartid = 0: User exit with code: 0.
```

图 1：app_print_backtrace

```
Shell >>> /bin$ app_errorline
into wait
hartid = 0, Application: /bin/app_errorline
hartid = 0, Application program entry point (virtual address): 0x0000000000010078
Going to hack the system by running privilege instructions.
Runtime error at user/app_errorline.c: 12
 asm volatile("csrw sscratch, 0");
Illegal instruction!
System is shutting down with exit code -1.
```

图 2：app_errorline

```
spike -p2 obj/riscv-pke /bin/shell
In m_start, hartid:0
In m_start, hartid:1
hartid = 0, User application is loading.
hartid = 0, Application: /bin/shell
hartid = 1, User application is loading.
hartid = 1, Application: obj/riscv-pke
hartid = 0, Application program entry point (virtual address): 0x00000000000100b0
RISCV-PKE Shell. Type 'leave' to leave.
Shell >>> /$
```

图 3：多核启动及运行

```
Shell >>> /bin$ app_sum_sequence
into wait
hartid = 0, Application: /bin/app_sum_sequence
hartid = 0, Application program entry point (virtual address): 0x00000000000100aa
handle_page_fault: 000000007fffdff8
handle_page_fault: 000000007fffcff8
handle_page_fault: 000000007fffbff8
handle_page_fault: 000000007fffaff8
handle_page_fault: 000000007fff9ff8
handle_page_fault: 000000007fff8ff8
handle_page_fault: 000000007fff7ff8
handle_page_fault: 000000007fff6ff8
handle_page_fault: 0000000000401000
this address: 0000000000401000 is not available!
hartid = 0: User exit with code: 1.
```

图 4：app_sum_sequence

```
Shell >>> /bin$ app_singlepageheap
into wait
hartid = 0, Application: /bin/app_singlepageheap
hartid = 0, Application program entry point (virtual address): 0x0000000000010094
hello, world!!!
hartid = 0: User exit with code: 0.
```

图 5：app_singlepageheap

```
Shell >>> /bin$ app_wait
into wait
Grandchild process end, flag = 2.
hartid = 0: User exit with code: 0.
hartid = 0, Application: /bin/app_wait
hartid = 0, Application program entry point (virtual address): 0x00000000000100b
0
into wait
Parent process end, flag = 0.
hartid = 0: User exit with code: 0.
Child process end, flag = 1.
hartid = 0: User exit with code: 0.
```

图 6：app_wait

```
Shell >>> /bin$ app_semaphore
into wait
hartid = 0, Application: /bin/app_semaphore
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8
Going to call sem.
Called sem.
Called sem.
Parent print 0
Child0 print 0
Child1 print 0
Parent print 1
Child0 print 1
Child1 print 1
Parent print 2
Child0 print 2
Child1 print 2
Parent print 3
Child0 print 3
Child1 print 3
Parent print 4
Child0 print 4
Child1 print 4
Parent print 5
Child0 print 5
Child1 print 5
Parent print 6
Child0 print 6
Child1 print 6
Parent print 7
Child0 print 7
Child1 print 7
Parent print 8
Child0 print 8
Child1 print 8
Parent print 9
hartid = 0: User exit with code: 0.
Child0 print 9
hartid = 0: User exit with code: 0.
```

图 7：app_semaphore

```
Shell >>> /bin$ app_cow
into wait
Child1 print 9
hartid = 0: User exit with code: 0.
hartid = 0, Application: /bin/app_cow
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8
the physical address of parent process heap is: 0000000087ead000
hartid = 0: User exit with code: 0.
the physical address of child process heap before copy on write is: 0000000087ea
d000
handle_page_fault: 0000000000400000
the physical address of child process heap after copy on write is: 0000000087e98
000
hartid = 0: User exit with code: 0.
```

图 8：app_cow

```
Shell >>> /$ cd /bin
Shell >>> /bin$ cd ..
Shell >>> /$ cd /bin
Shell >>> /bin$
```

图 9：cd

```
Shell >>> /bin$ ./app_pwd
into wait
hartid = 0, Application: /bin/app_pwd
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8

======== pwd command ========
cwd:/bin
hartid = 0: User exit with code: 0.
```

图 10：app_pwd

```
Shell >>> /bin$ ./app_mkdir /RAMDISK0/sub_dir
into wait
hartid = 0, Application: /bin/app_mkdir
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8

======== mkdir command ========
mkdir: /RAMDISK0/sub_dir
hartid = 0: User exit with code: 0.
```

图 11：app_mkdir

```
Shell >>> /bin$ app_touch /RAMDISK0/sub_dir/ramfile1
into wait
hartid = 0, Application: /bin/app_touch
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8

======== touch command ========
touch: /RAMDISK0/sub_dir/ramfile1
file descriptor fd: 0
hartid = 0: User exit with code: 0.
Shell >>> /bin$ ./app_touch /RAMDISK0/sub_dir/ramfile2
into wait
hartid = 0, Application: /bin/app_touch
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8

======== touch command ========
touch: /RAMDISK0/sub_dir/ramfile2
file descriptor fd: 0
hartid = 0: User exit with code: 0.
```

图 12：app_touch

```
Shell >>> /bin$ app_echo /RAMDISK0/sub_dir/ramfile1
into wait
hartid = 0, Application: /bin/app_echo
hartid = 0, Application program entry point (virtual address): 0x000000000000100b
0

======== echo command ========
echo: /RAMDISK0/sub_dir/ramfile1
file descriptor fd: 0
write content:
hello world
hartid = 0: User exit with code: 0.
```

图 13：app_echo

```
Shell >>> /bin$ ./app_cat ../RAMDISK0/sub_dir/ramfile1
into wait
hartid = 0, Application: /bin/app_cat
hartid = 0, Application program entry point (virtual address): 0x000000000001007
8

======== cat command ========
cat: /RAMDISK0/sub_dir/ramfile1
file descriptor fd: 0
read content:
hello world
hartid = 0: User exit with code: 0.
```

图 14：app_cat

```
Shell >>> /bin$ /bin/app_ls ../RAMDISK0/sub_dir
into wait
hartid = 0, Application: /bin/app_ls
hartid = 0, Application program entry point (virtual address): 0x00000000000100b
0
---------- ls command -----------
ls "/RAMDISK0/sub_dir":
[name]              [inode_num]
ramfile1            2
ramfile2            3
-----------------------------
hartid = 0: User exit with code: 0.
Shell >>> /bin$ /bin/app_ls /RAMDISK0
into wait
hartid = 0, Application: /bin/app_ls
hartid = 0, Application program entry point (virtual address): 0x00000000000100b
0
---------- ls command -----------
ls "/RAMDISK0":
[name]              [inode_num]
sub_dir             1
-----------------------------
hartid = 0: User exit with code: 0.
```

图 15：app_ls

由于上述绝大部分命令只需要将前面挑战实验修改过的代码移植至此即可，此处仅选取一个加分项进行简单叙述：保留并输出用户的命令历史记录。

由于此前已经实现了格式化读取用户输入，为了记录用户曾经输入的命令，只需要额外建立一个数组 log，一旦接收到用户命令即将其复制到数组中。log 数组的大小可以根据实际需求进行调整（例如 20），且可以使用 count 和 total 对历史命令的数量进行维护，当命令总数大于数组容量时，新命令可以覆盖旧命令，从而达到循环存储的目的。当用户当前输入为 history 时，按照从前向后的顺序取出数组中的元素并逐行打印即可。代码如下所示。

```
int main(int argc, char *argv[])
{
   ……
   char log[20][64];
   int count = 0, total = 0;
   strcpy(log[count], cmd);
   count = (count + 1) % 20;
   total++;
   ……
```

```
    else if (strcmp("history", cmd) == 0)
      {
        for (int k = (total - count) % 20; k < count; k++)
          printu("%s\n", log[j]);
        continue;
      }
  ……
```

最终效果如图 16 所示。

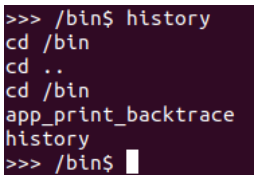

图 16：命令历史记录

1.3 实验调试及心得

本次实验是操作系统课设中最大的挑战，因为涉及到全部的挑战实验知识点，在合并时很容易因为不兼容的问题导致 Shell 报错。并且需要修改的代码量也相当大，需要反复调试才能输出正确结果，这里一个比较好用的技巧是在代码关键位置打印调试信息（比如在 wait 函数前后分别打印，从而判断进程的状态）。

这次课设是对上学期操作系统理论课的一次很好的补充，帮助我深入理解了内存、进程、线程、文件系统等核心功能在系统内核级上的代码实现技巧，让我从原先只会使用命令行 cmd 提高到能自己实现一个简易版的 Shell。在整个过程中，我也遇到了不少问题，例如在嵌入用户程序时内存未对齐（Misaligned AMO，如图 17 所示），以及分页时跳转 mepc 触发了 Instruction Page Fault 等等。

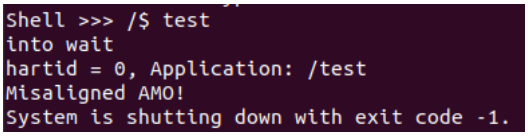

图 17：内存未对齐

后来发现，这种情况大概率是由于传入了一个空指针而报错，这也提醒我在写代码时一定要注意避免非法的地址访问。总体来说，本次课设丰富了我对大规模系统项目的调试经验，也提高了我对 RISCV 架构的理解。